Technische Universität Braunschweig, Informatik–Bericht 92–02 (1992)

TROLL *light*: A Core Language for Specifying Objects*

Stefan Conrad, Martin Gogolla, and Rudolf Herzig
Technische Universität Braunschweig, Informatik, Abt. Datenbanken
Postfach 3329, W-3300 Braunschweig, Germany
e-mail: {conrad|gogolla|herzig}@idb.cs.tu-bs.de

— December 1992 —

Abstract

TROLL *light* is a language for conceptual modeling of information systems. It is designed to describe the Universe of Discourse (UoD) as a system of concurrently existing and interacting objects. TROLL *light* objects have observable properties modeled by attributes, and the behavior of objects is described by events. Possible object observations may be restricted by constraints, whereas event occurrences may be restricted to specified life cycles. TROLL *light* objects are organized in an object hierarchy established by subobject relationships. Communication among objects is supported by event calling. Apart from introducing the various possibilities for the syntactical description of objects, we aim to describe how the state of an object community may be changed by event occurrences.

1 Introduction

In recent years formal specification techniques have been receiving more and more attention (see for example [CHJ86, TM87, Win90]). Formal specifications are used for supporting the program and system development process. Their design is an important step in the software development process, because formal specification leads to better software which can be proved to be correct with regard to the formal specification. Moreover, the development process for larger software can hopefully be shortened.

Once a correct formal specification of a system is given, it can be implemented. For implementation different methods are known: on the one hand transformation of the specification towards an executable specification (i.e., a program), or, on the other hand, direct implementation using an ordinary programming language. In both cases, the correctness of the transformation resp. the direct implementation must be proven with respect to the formal specification. All these aspects of software development are covered by the KorSo project[1].

There are several approaches to specification (see [Saa92]). For example: specification of functions (VDM, Z [BHL90]), abstract data types [EM85, EM90, among others], predicate logic and extensions like temporal and modal logic, semantic data models [HK87], and

*Work reported here has been supported by the German Ministry for Research and Technology (BMFT) under Grant No. 01 IS 203 D.

[1]KorSo stands for "KORrekte SOftware (= "correct software")

process specification (CCS [Mil80], CSP [Hoa85]), petri nets [Rei85]). But for different reasons, none of the above approaches seems to meet on their own all the requirements needed for the conceptual modeling of informations systems: The specification of functions and abstract data types does not adequately support the interactive nature of informations systems or persistent objects, predicate logic seems to be too general, semantic data models do not take into account the evolution of information systems, and process specifications do not reflect structural aspects.

The paradigm of object-orientation offers more natural ways to model systems and to develop modular software. Common catchwords used to describe the advantages of object-oriented modeling and programming are *encapsulation*, *inheritance* and *reusability*. The two areas of formal specification and object-orientation are brought together by object-oriented specification. In particular, object-oriented specification tries to take advantage of several specification techniques by combining their capabilities. For this extremely current research field semantic foundations of object-oriented specifications are given for instance in [EGS90, Mes90, SE91]. But in addition to semantic foundations, concrete specification languages for objects are needed as well. As far as we know OBLOG [SSE87, CSS89, SSG+91, SGG+91, SRGS91] was the first proposal. Based on experience with this language and on results achieved in the ESPRIT BRA IS-CORE the language TROLL [JSH91, JSHS91] was developed. Other specification languages for objects are ABEL [DO91], CMSL [Wie91], MONDEL [BBE+90], OS [Bre91], and Π [Gab91].

In the KorSo project our focus is on the development of reliable information systems. Therefore we were looking for a specification language to fulfill the following requirements: The language had to be simple enough to facilitate rapid prototyping as well as proving specification properties. On the other hand, the language had to be expressive enough to carry out some serious case studies.

Since we did not want to re-invent the wheel, we took the specification language TROLL as a basis for our language. A first comprehensive language description of TROLL appeared when our project started. It should be noted that the design of TROLL has been heavily influenced by a background in database design and conceptual modeling, especially by experience in the Entity-Relationship model [EGH+92]. The richness of concepts in TROLL allows us to model the Universe of Discourse (UoD) as adequately as possible. With this aim in mind even some partially redundant concepts are offered by TROLL. Thus it was necessary to evaluate which concepts are redundant and could be disregarded for our language purposes. Other concepts, like temporal logic for constraints, have been excluded for pragmatic reasons (so as to simplify prototyping and to make verification manageable). Finally, we obtained a language with a minimal number of concepts and hence called it TROLL *light*.

However, TROLL *light* is not just a subset of TROLL. Some details have been added or modified in order to round off TROLL *light*. This was necessary because we needed a clear and balanced semantic basis for our specification language. In particular we want to stress the fact that in TROLL *light* classes are understood as composite objects having the class extension as subobjects. Therefore in contrast to TROLL an extra notion of class is not needed in TROLL *light*. This leads to a more orthogonal use of object descriptions. Over and above that concepts like class attributes, meta-classes, or heterogeneous classes are inherent in TROLL *light* while they had to be introduced in TROLL by additional language features. Second TROLL *light* incorporates a query calculus providing a general declarative query facility for object-oriented databases. For instance, terms of this calculus may be used in object specifications to describe derivation rules for attributes, or to query object communities in an ad hoc manner.

Our specification language can only be a core language. It is not intended to be a comfortable specification vehicle. However, we kept an eye on maintaining the capability of expressing many concepts which we did not take into our language. By means of concepts included in our language, it should be possible to transform specifications of objects written in a more comfortable specification language into TROLL *light*.

The remainder of this paper is organized as follows. For the description of the properties of objects, data types are needed. Therefore, we briefly deal with data types in Sect. 2, before we introduce TROLL *light* in Sect. 3. Afterwards, in Sect. 4 we point out in more detail how the state of an object community is changed by event occurrences. Finally, in Sect. 5 we compare TROLL *light* with TROLL and semantic data models, and conclude with some remarks concerning future investigations.

2 Data Level

Today, the differences between data and objects are well-known and widely accepted [Bee90, GW91, SE91]: For example, the instance *5* of data sort `int` is a *universally known* abstraction. In contrast, an instance of object sort `employee` is an *application specific* abstraction. For the number *5* there exists a common natural representation by a printable value, while an employee can only find an artificial representation by an internal, non-visible object identifier (surrogate), which might be considered as a non-printable value of a non-lexical object type (NOLOT, [VB82]).

However, in some cases it is not quite clear, whether a real world entity should be modeled by a (complex) value or by an object. For example: consider the mathematical structure of a graph. A graph might be represented by the sets of nodes and edges, e.g., ({*1,2,3*}, {(*1,2*), (*1,3*)}), or it might be represented by an internal name. In the first case the graph is considered as a complex value, while in the second case it is considered as an object. The data sort `graph` might be given by `tuple(N:set(int), E:set(tuple(Start:int, End:int)))` in the first case, while in the second case the object sort `graph` will be associated with attributes like `N` : `graph` → `set(int)` and `E` : `graph` → `set(tuple(Start:int, End:int))`, and the assignment of values to an instance of `graph` is described by its *state*, which can alter in the course of time. So what is the appropriate representation? There are some criteria, which show the modeling as objects.

Object sharing: If a certain complex object should be referenced by other objects, it is much simpler and safer to reference it by a name instead of describing it with a complex value.

Updates on objects: Suppose that an edge should be removed from a graph. If, with respect to its identity, the modified graph should be considered as the old one, then it must be identified by a name instead of a complex value. At the same time, with object sharing update anomalies can be prevented as well.

Cyclic object modeling: Complex data sorts are built over atomic domains with the help of some type constructors, such as constructors for tuples and sets. Within such a constructive approach to types only finite hierarchical structures can be described. In contrast, object sorts allow for cyclic references by means of surrogates.

However, when none of the above criteria apply, it might be a better solution to represent a complex entity by a complex value [LR89]. This is because for the modeling as an object an additional naming mechanism must be provided.

Here, the language TROLL *light* is proposed for the specification of objects. Nevertheless, data types are needed to describe the properties of objects. In the literature, various styles for the description of data types are proposed. One of the most well-studied approaches is a descriptive one, for instance by means of equations [EM85, EGL89, EM90, Wir90]. The algebraic specification language SPECTRUM [BFG+92], which is also being developed within the KORSO project, is a good example of this.

Because we wish to concentrate on the specification of objects, and not on the specification of data types, standard and user-defined data types are assumed to be specified elsewhere. Consequently, the designer of TROLL *light* specifications only has to state which data types are to be used in an object description. However, some type constructors will still appear for the specification of attribute ranges. Basic notions of data types, sort expressions, and functions induced by sort expressions will be briefly described next. Complete formal definitions and further observations can be found in [GH91].

2.1 Data Signature

The syntax of a data signature is given by a set of data sorts together with a set of operations and predicates defined upon these sorts. We assume certain sorts (e.g., *nat*, *int*, *real*, *string*) and standard operations (e.g., $+_{nat} : nat \times nat \rightarrow nat$) and predicates (e.g., $\leq_{nat}$: $nat \times nat$) to be predefined. The semantics of a data signature associates a (possibly infinite) set of values with every sort name, and functions and relations with the operation and predicate names. We assume the semantics to be specified by axioms in first order predicate language or simply by equations. This interpretation will be fixed once and for all and will not change.

The predefined sorts as well as operations and predicates upon these sorts have the 'usual' semantics. For the interpretation of data sorts (and later object sorts) we suppose all corresponding sets to contain a special element $\perp$ denoting the *undefined* value or *null* value, because it is necessary to have such a value as a result of incorrect applications of operations, e.g., the division by zero always results in $\perp$.

Besides the standard sorts, a data signature may contain user-defined sorts, operations and predicates. With respect to user-defined sorts, atomic sorts (e.g., enumeration sorts such as `color` = {`green`, `yellow`, `red`}) are possible as well as arbitrary complex data sorts definable by *sort expressions*.

2.2 Sort Expressions

Each data sort (and later on each object sort) is considered as a simple sort expression. More complex sort expressions are built on the basis of five *sort constructors*. Let s denote a sort expression. Then $\mathbf{set}(s)$ is used to define finite sets over s,

Operation symbol	Arguments/Destination	Semantics
$\mathrm{CNT}_{set(s)}$	set(s) $\rightarrow$ *nat*	$\{\ \underline{s}_1,\ldots,\underline{s}_n\ \} \mapsto n$
$\mathrm{IN}_{set(s)}$	set(s) $\times$ s	$\{\ (\{\underline{s}_1,\ldots,\underline{s}_n\},\underline{s}) \mid \underline{s} \in \{\underline{s}_1,\ldots,\underline{s}_n\}\ \}$
$\mathrm{CNT}_{list(s)}$	list(s) $\rightarrow$ *nat*	$\langle\underline{s}_1,\ldots,\underline{s}_n\rangle \mapsto n$
$\mathrm{LTB}_{list(s)}$	list(s) $\rightarrow$ bag(s)	$\langle\underline{s}_1,\ldots,\underline{s}_n\rangle \mapsto \{\{\ \underline{s}_1,\ldots,\underline{s}_n\ \}\}$
$\mathrm{SEL}_{list(s)}$	list(s) $\times$ *nat* $\rightarrow$ s	$(\langle\underline{s}_1,\ldots,\underline{s}_n\rangle,i) \mapsto \begin{cases} \underline{s}_i & \text{if } 1 \leq i \leq n \\ \perp & \text{otherwise} \end{cases}$
$\mathrm{CNT}_{bag(s)}$	bag(s) $\rightarrow$ *nat*	$\{\{\ \underline{s}_1,\ldots,\underline{s}_n\ \}\} \mapsto n$
$\mathrm{BTS}_{bag(s)}$	bag(s) $\rightarrow$ set(s)	$\{\{\ \underline{s}_1,\ldots,\underline{s}_n\ \}\} \mapsto \{\ \underline{s}_1,\ldots,\underline{s}_n\ \}$
$\mathrm{a}^i_{tuple(a_1:s_1,\ldots,a_n:s_n)}$	tuple(a_1:s_1,...,a_n:s_n) $\rightarrow \mathrm{s}_i$	$(\underline{s}_1,\ldots,\underline{s}_n) \mapsto \underline{s}_i$
$\mathrm{PRJ}^i_{tuple(s_1,\ldots,s_n)}$	tuple(s_1,...,s_n) $\rightarrow \mathrm{s}_i$	
$\mathrm{a}^i_{union(a_1:s_1,\ldots,a_n:s_n)}$	union(a_1:s_1,...,a_n:s_n) $\rightarrow \mathrm{s}_i$	$\mathrm{a}_k{:}\underline{s}_k \mapsto \begin{cases} \underline{s}_k & \text{if } k = i \\ \perp & \text{otherwise} \end{cases}$

Figure 1: Operations and predicates induced by sort expressions

list(s) yields finite lists over s, and **bag**(s) denotes finite multisets (bags) over s. Let s_1,...,s_n be sort expressions and a_1,...,a_n be pairwise distinct tokens. Then Cartesian products can be constructed by **tuple**($[a_1{:}]s_1$,...,$[a_n{:}]s_n$). For example, a complex data sort `graph` could be constructed by `TUPLE(N:nodes, E:edges)` in which `nodes` is given by `SET(int)`, `edges` by `SET(TUPLE(Start:int, End:int))`. For instance, ({*1,2,3*}, {(*1,2*), (*1,3*)}) is a proper value of sort `graph`. Please note that tokens are optional in tuple expressions. Finally a disjoint union constructor is provided for TROLL *light*. **union**($a_1{:}s_1$,...,$a_n{:}s_n$) allows the modeling of alternatives. In contrast to tuple expressions, tokens are indispensable for union expressions. The semantics associates union($a_1{:}s_1$,...,$a_n{:}s_n$) with the disjoint union of all values found in the interpretations of s_1 through s_n. To achieve disjointness in the interpretation of union sorts, even in the case where some sort expression s appears more than once in $s_1,\ldots,s_n$, all values must be prefixed by the corresponding tokens a_i. For example, a data sort `address` could be given by `UNION(BA:business_address, PA:private_address)`, where `business_address` might be given by `TUPLE(Office:string, City:string)` and `private_address` by `TUPLE(Num:nat, Street:string, City:string)`. Then *BA:("Computer Center", "Paris")* or *PA:(13, "5th Avenue", "New York")* are proper values of sort `address`.

2.3 Operations and predicates induced by sort expressions

There are numerous operations and predicates associated with sort constructors. For example, CNT counts the elements in a set, list or bag, IN represents the “element of” relation, and there are operations selecting components of a tuple or alternatives of a union (see Fig. 1), and many other functions and predicates. For details see [GH91].

3 Object Specification

Object descriptions are the basic building blocks of the object-based specification of systems. Object descriptions are not the objects in a system, but they are descriptions of their static and dynamic properties. In both TROLL and TROLL *light* an object description is called a *template*. The structure of templates is given as follows:

```
TEMPLATE <name>
  DATA TYPES   data types used in current template definition
  TEMPLATES    templates used in current template definition
  SUBOBJECTS   subobject names local to current object²
  ATTRIBUTES   observable properties of current object
  EVENTS       events possibly occurring in current object's life cycle
  CONSTRAINTS  restricting conditions on object states
  VALUATION    effect of events on attributes
  DERIVATION   derivation rules for attributes
  INTERACTION  synchronization of events in different objects
  BEHAVIOR     description of object behavior by event-driven sequential machines
END TEMPLATE;
```

The following subsections provide a step-by-step introduction to the structure of templates.

3.1 Simple Objects

Let us assume that an object of type author is to be modeled. For every author the name, the date of birth, and the number of books sold by year have to be stored. An author may change his name only once in his life. An appropriate TROLL *light* specification would be:

```
TEMPLATE Author
  DATA TYPES   String, Date, Nat;
  ATTRIBUTES   Name:string; DateOfBirth:date; SoldBooks(Year:nat):nat;
  EVENTS       BIRTH create(Name:string, DateOfBirth:date);
                     changename(NewName:string);
                     storesoldbooks(Year:nat, Number:nat);
               DEATH destroy;
  VALUATION    [create(N,D)] Name=N, DateOfBirth=D;
               [changename(N)] Name=N;
               [storesoldbooks(Y,NR)] SoldBooks(Y)=NR;
  BEHAVIOR     PROCESS authorlife1 =
                 ( storesoldbooks -> authorlife1 |
                   changename -> authorlife2 |
                   destroy -> POSTMORTEM );
               PROCESS authorlife2 =
                 ( storesoldbooks -> authorlife2 |
                   destroy -> POSTMORTEM );
               ( create -> authorlife1 );
END TEMPLATE;
```

[2] "Current object" means an instance of the current template.

Data type specifications result in data sorts, whereas object specifications (templates) come along with object sorts. We use the same names for data types and the corresponding data sorts or for templates and the corresponding object sorts, but we require data types and templates to start with a capital letter whereas the sorts are set to small letters. Thus `Author` denotes a template while `author` denotes the corresponding object sort.

As mentioned in Sect. 2, **data types** are assumed to be specified somewhere else. By the `DATA TYPE` section the signature of data types is made known to the current template. For example, referring to Nat means that the data sort *nat*, operations like $+_{nat} : nat \times nat \to nat$, and predicates like $\leq_{nat}$: $nat \times nat$ are visible in the current template definition.

Attributes denote observable *properties* of objects. They are specified in the `ATTRIBUTE` section of a template by $a\,[\,\texttt{(}[p_1\!:]s_1, \ldots, [p_n\!:]s_n\texttt{)}\,]\texttt{:}d$, where a is an attribute name generator, d is a sort expression determining the range of an attribute, and $s_1, \ldots, s_n$ ($n \geq 0$) denote optional parameter sorts (data or object sorts). To stress the meaning of parameter sorts, optional parameter names p_i might be added.

The sort expression d may be built over both data sorts and object sorts by applying the sort constructors presented in Sect. 2. Thereby, arbitrary *complex* attributes can be specified, e.g., data-valued, object-valued, multi-valued, composite, alternative attributes, and so on. The interpretation of all sort expressions contains the undefined element $\perp$. Hence all attributes are *optional* by default.

Attribute names may be provided with parameters. For example, by the declaration `SoldBooks(Year:nat):nat` a possibly infinite number of attribute names like *SoldBooks(1992)* is introduced. We assume that in a given state of an object only a finite number of attributes will take defined values in such a way that only these attributes have to be stored. It is of interest to note that a parameterized attribute $a(s_1, \ldots, s_n) : s$ could also be viewed as an attribute $a : \mathrm{set}(\mathrm{tuple}(s_1, \ldots, s_n, s))$, but clearly the formerly inherent functional dependency would have to be described by an explicit constraint.

Incidents possibly appearing in an object's life are modeled by **events**. Events are specified in the `EVENT` section of a template by $e[\texttt{(}[p_1\!:]s_1, \ldots, [p_n\!:]s_n\texttt{)}]$, where e denotes an event generator and $s_1, \ldots, s_n$ ($n \geq 0$) represent optional parameter sort expressions. To underline the meaning of parameters, optional parameter names p_i may be added.

Event parameters are used to define the effects of an event occurrence on the current state of an object (see the explanation of the `VALUATION` section below), or they are used to describe the transmission of data during communications (see the explanation of the `INTERACTION` section below). Special events in an object's life cycle are the `BIRTH` and `DEATH` events with which an object's life is started or ended. Several birth or death events may be specified for the same object. A template may have no death event, but we require a template to have at least one birth event.

The effect of events on attribute values is specified in the `VALUATION` section of a template definition by **valuation rules**, having the following form.

$$[\texttt{\{}\mathit{precondition}\texttt{\}}]\ \texttt{[}\mathit{event\_descr}\texttt{]}\ \mathit{attr\_term}\ \texttt{=}\ \mathit{term}$$

Such a rule says that immediately after an event occurrence belonging to the event descriptor *event_descr*, the attribute given by the attribute term *attr_term* has the value denoted by destination term *term*[3]. The applicability of such a rule may be restricted by a *precondition*. This precondition as well as both terms is evaluated before the event occurrence. Thereby all these items may have free variables which, however, must appear in the event descriptor. An *event descriptor* consists of an event generator and a possibly empty list of variables representing the parameters of an event. By a concrete occurrence of such an event, these variables are instantiated with the actual event parameters so that the precondition and the other terms can be evaluated.

To summarize we can say that a valuation is a proposition stating that after the occurrence of a certain event some attributes shall have some special values. The following frame rule is assumed: attributes which are not caught by a suitable valuation formula of a given event remain unchanged. Before the birth event, all attributes of an object are assumed to be undefined. Thus if the event in question is a birth event, some attributes may remain undefined.

In our example the birth event `create` has effects on more than one attribute. In that case a comma can be used as a short-hand notation to combine several equations. It is important to note that an attribute can only be affected by local events, i.e., events which are specified in the same template in which the attribute is specified. The one and only exception concerns object-valued attributes, where *implicit valuation* can be produced by non-local events (see Sect. 4).

In the `BEHAVIOR` section possible life cycles of objects are specified. Using an abstract characterization, **behavior patterns** are described by an event-driven sequential machine, which will be called *o-machine* (o for object) in the sequel. This machine has an initial state (to which we need not refer in specifications), a final state `POSTMORTEM`, and a finite number of life states (cf. [SM92]). Transitions between o-machine states are stated by a partial function which maps a tuple consisting of actual state, event descriptor, and optional precondition to a consecutive state.

To achieve a compact notation within the `BEHAVIOR` section such an o-machine is represented by process descriptions. By process descriptions, *event sequences*, which an object must go through, can be specified as well as *event dependent branchings*. Note that event sequences always lead back into processes, and that event alternatives, which produce the same o-machine state transitions, can be listed separated by commas.

The possibility of providing every state transition with a precondition allows for *guarded events* (cf. CSP [Hoa85]). In Fig. 2 the behavior of authors is visualized by the corresponding o-machine representation (another example appears below). Please note that the special life states, the initial state and the final state `POSTMORTEM`, are not explicitly drawn in such object behavior diagrams. To be precise arrows coming from the blue depart from the initial state, and arrows directed to the blue lead to `POSTMORTEM`. Semantically the initial and the final state coincide. Within an object description the behavior section may be missing. In that case life cycles are unrestricted, i.e., then it is only required for life cycles to start with a birth event and possibly to end with a death event.

[3] `[`*event_descr*`]` can also be interpreted as a next operator of some temporal/modal logic.

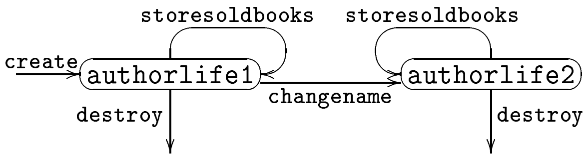


Figure 2: Behavior of authors

3.2 Composite Objects

In the last section only the template of single author objects has been described. In order to combine several authors in a higher-level object, *classes* are usually introduced in object-oriented approaches. Here, TROLL *light* does not support an explicit class concept. Classes are viewed as composite objects instead, and therefore classes have to be described by templates as already mentioned in the last section. However, the means of describing the relationship between a container object and the contained objects must be added. This is done by introducing subobject relationships denoting (exclusive) *part-of* relationships. The following example gives the shape of container objects for authors.

```
TEMPLATE Authorclass
  DATA TYPES   String, Date, Nat;
  TEMPLATES    Author;
  SUBOBJECTS   Authors(No:nat):author;
  ATTRIBUTES   NumberOfAuthors:nat;
  EVENTS       BIRTH createclass;
                     addobject(No:nat, Name:string, DateOfBirth:date);
                     removeobject(No:nat);
               DEATH destroyclass;
  VALUATION    [createclass] NumberOfAuthors=0;
               [addobject] NumberOfAuthors=NumberOfAuthors+1;
               [removeobject] NumberOfAuthors=NumberOfAuthors-1;
  INTERACTION  addobject(N,S,D) >> Authors(N).create(S,D);
               removeobject(N) >> Authors(N).destroy;
END TEMPLATE;
```

Within the `TEMPLATES` section, other existing **templates** can be made known to the current template. For example, referring to `Author` means that the object sort `author`, and the attributes and the event generators of `Author` are visible in `Authorclass`.

An object of sort `authorclass` will hold many author objects as private components or **subobjects**. In order to be able to distinguish between several authors from a logical point of view an *identification mechanism* is needed. One way to install such a mechanism would be to assign a unique name for each subobject, e.g., `MyAuthor`, `YourAuthor`, ... Indeed, such a name allocation could be expressed in TROLL *light* by the declaration

```
SUBOBJECTS  MyAuthor:author; YourAuthor:author; ...;
```

But clearly for lots of authors such a procedure is not practicable. A solution is given by parametrized subobject constructors as shown in the example. As with parametrized attributes, a possibly infinite number of logical subobject names like *Authors(42)* can be defined by the subobject name declaration for authors, but not the author objects themselves! The declaration only states that in context of `Authorclass`, author objects are identified by a natural number. Logical object names are linked with actual objects (or rather abstract object identifiers) when a parent object calls for the birth of a subobject, which is described by interaction rules. Since a birth event can only happen once in a life of an object, it therefore follows that a subobject can never be part of two superobjects.

Interaction rules are specified in the `INTERACTION` part of the template definition. During the lifetime of an author class there will be events concerning the insertion or deletion of authors. In `Authorclass`, the insertion of an author should always be combined with the creation of a new author object. In the template this is expressed by `addobject(N,S,D) >> Authors(N).create(S,D)`, where `N` denotes a natural number, `S` a string, and `D` a date. The general event calling scheme is expressed as

[{*precondition*}] [*src_obj_term* .]*src_event_descr* `>>` [*dest_obj_term* .]*dest_event_term* .

Such a rule states that whenever an event belonging to the source event descriptor *src_event_descr* occurs in the source object denoted by *src_obj_term*, and the *precondition* holds, then the event denoted by the destination event term *dest_event_term* must occur simultaneously in the destination object denoted by *dest_obj_term*. This form of event calling was introduced in [SEC90].

If one of the object terms is missing, then the corresponding event descriptor (resp. event term) refers to the current object. The destination object term, the destination event term, and the precondition may only have free variables which occur in the source object term or the source event descriptor. By the occurrence of an event in some object matching the source object term and the source event descriptor these free variables become bound so that the precondition can be checked, and the destination object term and the destination event term are completely instanciated.

Sometimes we may wish that the occurrence of some special event causes the occurrence of a whole set of other events. Especially, if the set of corresponding objects depends on the state of the specified system we can write a set-valued destination object term.

In almost all cases the objects to be called for must be defined. The one and only exception is when a parent object calls for the creation of a subobject. In this case it is even a requisite that the destination object is not alive at that moment. For our example this means that the subobject name denoted by `Authors(N)` is going to be associated with a new author object. Strictly speaking the logical subobject name is associated with an abstract object identifier returned by the creation of an author object. In any case this association is fixed for the life of an object. It is dissolved only when the object is deleted by the occurrence of a death event. Of course, the name is then free to be used again for a new object to be created.

In the next subsection we extend our example by the definition of a book and book class template, and we show how books and authors can be used by two libraries. At the same time some further concepts of TROLL *light* are introduced.

3.3 More Sophisticated Concepts of TROLL *light*

First the example used hitherto is extended by the definition of a book template. A list of authors and a title may be given for each book.

```
TEMPLATE Book
  DATA TYPES   String;
  TEMPLATES    Author;
  ATTRIBUTES   Authors:LIST(author); Title:string;
  EVENTS       BIRTH create(Authors:LIST(author), Title:string);
               DEATH destroy;
  CONSTRAINTS  CNT(Authors)>0;
  VALUATION    [create(AL,T)] Authors=AL, Title=T;
END TEMPLATE;
```

Subobject relationships yield a strict embedding of objects into superobjects. A more dynamic form of object assignment is given by *object-valued attributes*. These may be used especially for *object sharing*. In our example books have a list-valued attribute `Authors`. Please note that in contrast to subobjects, which can be created within a parent object by birth *event calling*, objects denoted by object-valued attributes are assigned by *valuation rules*, which means that such objects must already exist at the time they are assigned.

Possible object states can be restricted by explicit **constraints**, which are specified in the `CONSTRAINTS` section of a template. Constraints are given by formulas without free variables. In our example books are required to have at least one author.

A book class template can be defined in the same way as we did for the authors.

```
TEMPLATE Bookclass
  DATA TYPES   String, Nat;
  TEMPLATES    Author, Book;
  SUBOBJECTS   Books(Author:author, Title:string):book;
  ATTRIBUTES   DERIVED NumberOfBooks:nat;
  EVENTS       BIRTH createclass;
                     addobject(Authors:LIST(author), Title:string);
                     removeobject(Author:author, Title:string);
               DEATH destroyclass;
  DERIVATION   NumberOfBooks = CNT(Books);
  INTERACTION  addobject(AL,T) >> Books(SEL(AL,1),T).create(AL,T);
               removeobject(A,T) >> Books(A,T).destroy;
END TEMPLATE;
```

Books are identified by the author placed first in the author list and the title. Thus an object-valued parameter is used for identification. In consequence, when a book is to be created, the associated author must already exist, and when an author is to be deleted, then all book objects whose logical names are built upon this author, have to be removed, which can be viewed as a matter of *implicit death event calling* (see Sect. 4). So it should be emphasized that the use of object-valued keys leads to an *existence dependency* of the object which is to be identified from the object which is used for identification.

In the example a *derived attribute* is introduced. `NumberOfBooks` returns the number of books which are currently part of a book class. It may be compared with the attribute `NumberOfAuthors` of author class. But while `NumberOfAuthors` is set by explicit valuation rules, the same task is achieved in book class by a single **derivation rule**. Here it should be noticed that each parametrized attribute or subobject name generator defined for a template t induces a corresponding set-valued operation on the corresponding object sort, which returns the finite set of all current tuples built from parameter values and defined destination values. For example, `Books` is a term of sort `SET(TUPLE(author, string, book))`. For a subobject name generator s this term is evaluated as the set of all life objects identified via s together with their keys. The aggregate function `CNT` is used to count these objects.

The shape of documents is described in the following template. Documents are going to be subobjects of libraries. It is assumed that every document in a library is associated with a fixed book. Of course there may be several documents which are copies of the same book. Every document has a status denoting whether the document is `available`, `borrowed` or `reserved`. These three values constitute the possible values of a data type `docstatus`.

```
TEMPLATE Document
  DATA TYPES   Docstatus;
  TEMPLATES    Book;
  ATTRIBUTES   Book:book; Status:docstatus;
  EVENTS       BIRTH create(Book:book);
                     borrow; return; reserve;
               DEATH destroy;
  CONSTRAINTS  DEF(Book);
  VALUATION    [create(B)] Book=B, Status=available;
               [borrow]  Status=borrowed;
               { Status<>reserved } [return] Status=available;
               [reserve] Status=reserved;
  BEHAVIOR     PROCESS documentlife =
                 ( borrow -> borroweddocumentlife |
                   destroy -> POSTMORTEM );
               PROCESS borroweddocumentlife =
                 ( return -> documentlife |
                   reserve -> return -> borrow -> borroweddocumentlife );
               ( create -> documentlife );
END TEMPLATE;
```

The predicate `DEF` in the `CONSTRAINTS` section says that the value of `Book` must take a defined value in all life states. Documents show a specific behavior which is set out in the `BEHAVIOR` part. Borrowed documents can be returned, or a reservation for a borrowed book may appear. If a document has been reserved, then the document must be returned first, and afterwards it has to be borrowed again; for instance, by the person who made the reservation. In Fig. 3 the corresponding o-machine representation is given. Both event sequences, e.g., `reserve -> return -> borrow`, and event dependent branchings, e.g., `reserve` ... `| return` ... are present.

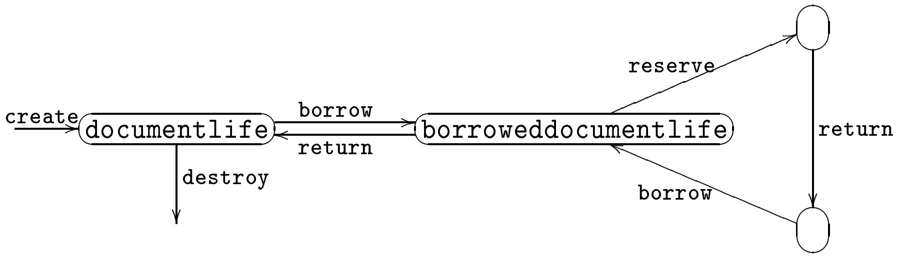


Figure 3: Behavior of documents

The template gives an example for *conditional valuation*, stating that only if a document is not reserved, then the event `return` can make a document to an available one. Otherwise the status remains unchanged.

Documents are subobjects of libraries. In a library each document is identified by a unique document number. In order to refer to authors and books every library uses an object of sort `authorclass` and an object of sort `bookclass`.

```
TEMPLATE Library
  DATA TYPES   String, Date, Nat;
  TEMPLATES    Author, Authorclass, Book, Bookclass, Document;
  SUBOBJECTS   Documents(DocNo:string):document;
  ATTRIBUTES   AuthorIndex:authorclass; BookIndex:bookclass;
  EVENTS       BIRTH create(AuthorIndex:authorclass, BookIndex:bookclass);
                     enterauthor(No:nat, Name:string, DateOfBirth:date);
                     changeauthorname(Author:author, NewName:string);
                     enterbook(Authors:LIST(author), Title:string);
                     enterdocument(DocNo:string, Book:book);
                     borrowdoc(DocNo:string);
                     reservedoc(DocNo:string);
                     returndoc(DocNo:string);
  CONSTRAINTS  DEF(AuthorIndex); DEF(BookIndex);
               CNT(Documents)<100000;
  VALUATION    [create(A,B)] AuthorIndex=A, BookIndex=B;
  INTERACTION  enterauthor(N,S,D) >> AuthorIndex.addobject(N,S,D);
               changeauthorname(A,N) >> A.changename(N);
               enterbook(AL,T) >> BookIndex.addobject(AL,T);
               enterdocument(D,B) >> Documents(D).create(B);
               { Status(Documents(D))<>borrowed }
                   borrowdoc(D) >> Documents(D).borrow;
               reservedoc(D) >> Documents(D).reserve;
               returndoc(D) >> Documents(D).return;
  BEHAVIOR     PROCESS liblife =
                 ( enterauthor, changeauthorname, enterbook,
                   enterdocument, borrowdoc,
                   { Status(Documents(D))=borrowed } reservedoc(D),
```

```
                  returndoc -> liblife )
              ( create -> liblife );
END TEMPLATE;
```

Since libraries refer to both internal objects given by subobjects and external objects given by object-valued attributes, the template has a rather large `INTERACTION` part. Event calling requires that objects to be called for are given in some way. Subobject name generators have been used in this way to synchronize parent object and subobjects. But as shown in `library`, other object-valued terms are also possible. For `enterauthor` the object-valued attribute `AuthorIndex` is used for event calling, and for `changeauthorname` an object-valued event parameter is employed.

The template gives an example for *conditional event calling*. Even if the event `borrowdoc` is not feasible, since the desired document is already borrowed, the event `borrowdoc` may appear in the life cycle of a library indicating a nil request for some documents.

The template also gives an example of a *guarded event*. The event `reservedoc` for a given document may only occur if the document is being borrowed. Please note that here the condition appears to be superfluous since it is already guaranteed by the behavior section of `document`.

The final template can be viewed as the specification of a very small database (or object base) schema. In a database of sort `libraryworld` two libraries called `LibA` and `LibB` manage a large number of documents. To this end they have common access to an `AuthorIndex` of sort `authorclass` and a `BookIndex` of sort `bookclass`.

```
TEMPLATE Libraryworld
  TEMPLATES    Authorclass, Bookclass, Library;
  SUBOBJECTS   AuthorIndex:authorclass; BookIndex:bookclass;
               LibA:library; LibB:library;
  EVENTS       BIRTH createworld;
                     createindexes;
                     createlibraries;
  INTERACTION  createindexes   >> AuthorIndex.create, BookIndex.create;
               createlibraries >> LibA.create(AuthorIndex, BookIndex),
                                  LibB.create(AuthorIndex, BookIndex);
  BEHAVIOR     ( createworld -> createindexes ->
                 createlibraries -> () );
END TEMPLATE;
```

The structure of an instance of sort `libraryworld` is set out in Fig. 4. The diagram is very similar to the object diagrams presented in [KS91]. Objects are represented by rectangles. The corresponding object sort is noted in the upper left-hand corner of a rectangle. Subobject relationships are illustrated by embedding rectangles in bigger rectangles. Since the logical name of an object is always defined within a superobject, it appears outside the related rectangle. Dashed lines represent object relationships established by object-valued attributes.

An instance of sort `libraryworld` might be viewed as a *schema object*. If we extend our view of a schema object to all its subobjects, this instance denotes an *object community*.

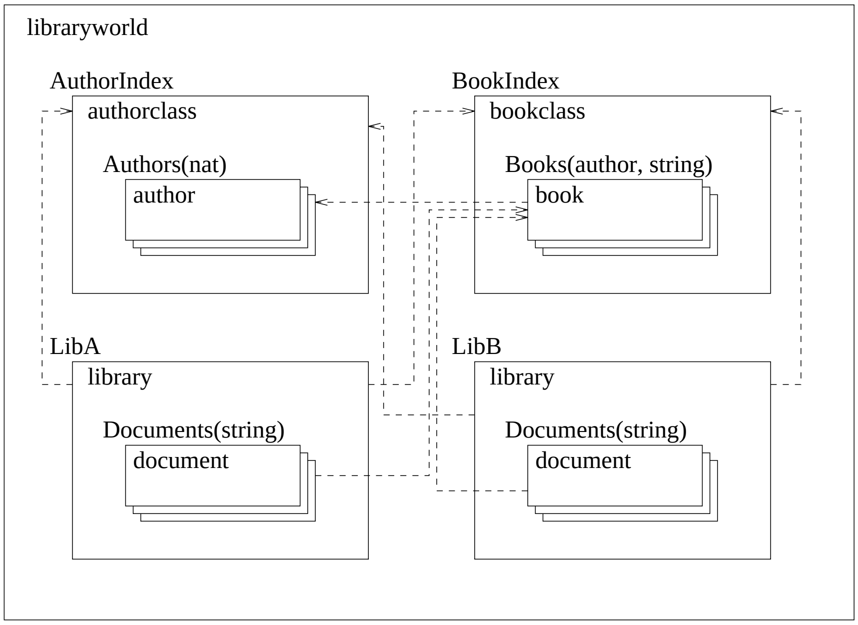


Figure 4: Structure of an object of sort `libraryworld`

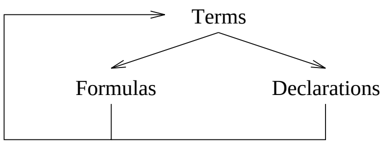
Subobject relationships impose a tree structure for an object community with the schema object as the root object. Further references, which go beyond the tree structure, can be provided by object-valued attributes. An object community o is called *closed* if all object references are directed to nodes belonging to o. The schema object has no logical name (just as until now nobody had the idea of assigning a name to our cosmos), which is in accord with the **principle of object existence** adopted for TROLL *light*:

> *Every live object must be the subobject of a higher object. Otherwise the very notion of object does not make sense, because an object can only be discerned as an object by a higher object.*

Even the schema object is supposed to be subobject of another object. In particular, this object might be still to be specified (for example, the database management system). Obviously, the birth event of an object of template `libraryworld` will be called by such a superobject, and this object will provide a logical name for the root object.

3.4 Query Calculus

TROLL *light* incorporates a query calculus which is an adaptation of a query calculus proposed for extended Entity-Relationship schemas [GH91]. This calculus has a non-hierarchical structure given by



which means that terms may be used to build formulas and declarations, and both formulas and declarations may appear in the construction of terms. The kernel of this calculus is given by bag-valued terms of the kind `SELECT` τ `FROM` δ `WHERE` φ in which τ is an arbitrary term with possibly free variables, δ is a declaration binding one or more free variables to finite domains given by set-valued terms, and φ is a qualifying formula. The calculus is as orthogonal as possible by allowing for arbitrary terms to be used in the target, declaration, and qualification clause.

We already used closed formulas as constraints in templates, and terms appeared frequently in templates such as to describe the destination value in valuation rules, to express derivation rules for attributes, or to compute the objects envolved in interactions. Besides that the query calculus can also be used to formulate ad hoc queries against object communities. It is a characteristic property of TROLL *light* object communities that the starting point of queries is always an object rather than a schema.

For example, the query "Give me the document numbers of documents with more than two authors" makes only sense in context of an object of sort `library`.

```
SELECT PRJ(d,1)
FROM   d IN Documents
WHERE  CNT(Authors(Book(PRJ(d,2)))) > 2;
```

Recall that `Documents` is a term of sort `SET(TUPLE(string,document))` so that projections must be applied to the variable `d` in order to refer to the first or second component.

In contrast the following query starts from an object of sort `libraryworld`: "Give me the books which exist in `LibA` but not in `LibB`."

```
BTS(SELECT Book(PRJ(d1,2))
    FROM   d1 IN Documents(LibA)
    WHERE  NOT EXISTS (d2 IN Documents(LibB))
                Book(PRJ(d2,2))=Book(PRJ(d1,2)))
```

Note that in general select-terms are evaluated to true bags rather than sets. Take for example two documents in `LibA` being copies of the same book which does not exist in `LibB`. Then the corresponding book appears twice in the result of the select term. However, duplicates can be removed by converting a bag to a set which is done by the aggregate function `BTS`.

We do not want to go into further details here. A detailed description of the adaptation of the calculus of [GH91] shall appear in a future paper.

4 State Transitions

The state of an object is described by the values assigned to its attributes, the objects assigned to its subobject names, and the state of its o-machine. The state of an object is called *consistent* if all constraints specified in the `CONSTRAINTS` section are fulfilled. The state of an object community is called consistent if the states of all its members are consistent.

Let us assume a closed object community to be given (for example by an instance of sort `libraryworld`). Reducing our view of objects to subobject relationships, an object community is a tree with the schema object as its root. The state of the attributes and the o-machine may be viewed as node information. If an event occurs to a member of this community, this will lead under certain circumstances to a state transition. Hence only state transitions between consistent states are possible.

The following points must be observed. The application of interaction rules may invoke further events for other objects. Thereby the object community tree may be altered by the occurrence of birth and death events. Birth events cause new nodes to be added, while death events cause nodes (or even whole parts) of the tree to be removed. With respect to attributes, the node information may be modified by applying explicit or implicit valuation rules, and the o-machine states are modified by applying state transition rules as described in the `BEHAVIOR` section of a template.

To compute a state transition, all objects affected by events must be determined. Firstly explicit interaction rules expressing event calling for live objects have to be evaluated. Here only one event may be called within a single object at a time (local non-concurrency).

Since a live object, which has to die by the occurrence of a death event, may have subobjects, or of this object is referenced by other objects via object-valued attributes, the effects of death events can be far reaching. This means that if a parent object is to be removed, then all its subobjects must be removed at the same time. If no explicit interaction rule exists for that case, this only can be viewed as a matter of *implicit death event calling* (enforced death). If a reference to an object exists, which is to be deleted, then the reference must be cancelled. To this end we must be careful in the case where an object-valued key is affected. Since the logical subobject name will collapse, the subobject must be deleted, which again must be viewed as a matter of implicit death event calling, unless a death event occurs explicitly. Clearly implicit death events and life events are not permitted to occur simultaneously.

Finally birth events must be considered. Birth event calling can only be directed to unborn (undefined) objects outside the object community. For every subobject name to be associated with a new subobject an unborn object of appropriate sort must be assigned. Thereby the birth event for this object is triggered. Here we must stress the fact that it is not allowed to create two subobjects under the same logical name.

In this way all objects directly affected by actions are fixed. The state transition is performed by updating the tree structure (install or cancel subobject relationships) and updating the node information (o-machine state and attributes). For each (explicit) event occurrence a suitable state transition must be predefined with respect to the corresponding o-machine. In principle, non-determinism may arise. Attributes are primarily updated by applying valuation rules. Here non-determinism is possible too. Secondly implicit valuation may appear when references to dying objects are to be cancelled. In any case, implicit valuation and explicit valuation must not contradict each other, i.e., values to be assigned by explicit valuation rules must not contain references to dying objects.

5 Conclusions

5.1 Beyond TROLL *light*

The language TROLL *light* may be used to describe structural and dynamic aspects of objects. It is influenced by semantic data models (specification of structure and states), abstract data types (specification of functionality), and process models (specification of dynamic aspects). Nevertheless, since we focus on database design, structural modeling in the sense of semantic data models is of major importance. With respect to structure TROLL *light* provides two modeling primitives, namely attributes and subobject relationships.

Attributes express observable properties of objects. Therefore attribute domains are allowed to be of any complexity supporting the modeling of complex values as required in [LR89].

Subobject relationships can be used to organize objects into an object hierarchy. Although this kind of object organization is well-known from file management, most data models choose another way of object structuring which is class-oriented. TROLL *light* does not acknowledge an explicit class concepts, but classes are treated as "class objects" and "class members" as subobjects instead. This has the advantage that additional concepts like class attributes, meta-classes, or heterogeneous classes are already inherent to TROLL *light*. In the field of object-oriented deductive databases a similar view to objects has been adopted by F-Logic [KL89] and LOCO [LVVS90], for example.

Semantic data models should enable the user "to think of data in ways that correlate more directly to how data arise in the world" [HK87]. Therefore a data model should support conceptually different views to objects as there might be relationship objects, role objects, interface objects, etc. TROLL *light* adopts a uniform view of objects so that a language like TROLL [JSHS91] seems to be much more appropriate for conceptual modeling than TROLL *light*. TROLL offers several additional constructs for conceptual modeling, e.g. generalization, roles, arbitrary relationships. Here it becomes clear that TROLL *light* is more implementation-oriented than TROLL. That is why TROLL *light* might be used as a kind of target language for logical design, since it should be possible to faithfully simulate missing concepts in TROLL *light*.

Semantic data models often neglect object dynamics by limiting themselves to generic updates. In contrast updates are subject to specification in TROLL and TROLL *light*. Precisely speaking object dynamics is addressed by the specification of events and behavior patterns. In contrast, TROLL supports the description of object behavior by permissions, obligations, process patterns, and commitment in a more declarative way. Especially the use of temporal logic allows a rather abstract (and semantically loose) specification of behavior. TROLL *light* sticks to an operational description by means of object machines which corresponds more or less to the process patterns of TROLL. Clearly, a temporal formula claiming that an author changes his name only once in his life is nearer at the UoD than coding it into states of an object-machine. The latter may be result of a refinement step which again underlines the position of TROLL *light* as a low-level specification language being nearer to implementation.

Communication is addressed in TROLL *light* by event calling demanding the synchronization of events. But here TROLL and TROLL *light* represent the same level of abstraction.

Summarizing we should emphasize that TROLL *light* offers a clear framework to study aspects which are relevant for our project which aims at the development of correct software. Some of these aspects are briefly discussed in last subsection. After having solved all the problems appearing in a simplified context (here TROLL *light*) one might be able to investigate the more complex issues provided by TROLL, for example.

5.2 Future Work

After we have now fixed the language, a lot of work remains to be done. Special topics for future investigations will be

- formal semantics of TROLL *light*,
- validation,
- verification, and
- an integrated development environment.

In this paper we have presented TROLL *light* only in an informal way and have not gone into its *formal semantics*. We are working on a computational model which gives an operational semantics for TROLL *light*. Of course, this semantics may be adequate for prototyping, for other purposes, however, we need other semantic approaches. Here the entity algebra approach of [Reg90] or concurrent term rewriting [Mes90, Mes92] may be suitable tools.

Validation is another aspect of our future studies. This is due to the fact that formal specifications cannot be proved to be correct with regard to some informal description of the system to be specified. Only validation can help to ensure that a formally specified system behaves as required. Animation of specifications seems to be an important way to support validation in an efficient manner. Therefore, we plan to develop and implement a prototyping system for TROLL *light* specifications supporting specification designers in order to animate object communities fulfilling their specifications.

Another aspect we intend to work on is *verification*. Verification is needed to prove properties of specifications. Writing all intended properties of some objects into their template specification seems to be unrealistic, because in this way specifications would become larger and larger and finally nobody would be able to read and understand such specifications. Furthermore, a lot of specified properties would be redundant. Therefore, it seems better to write "minimal" specifications which only include necessary properties. Derivable properties should be formulated outside the specification (within another language), but must be stored together with the specification. For these properties a calculus together with a deduction system is needed in order to express such properties and to reason about them. Finally, we also need tools to support the proving of specification properties.

A special aspect of validation and verification is *consistency*. Animation of objects, as well as proving properties of specifications, requires consistent specifications. Therefore, requirements for specifications to be consistent have to be worked out. After this, the checking of such conditions must be done automatically as far as possible. Tools have to be fashioned to support the specification designer in developing consistent specifications.

We also wish to design an *integrated development environment* for our specification language. In principle, this environment will support all phases of software development: information analysis, formal specification, animation, verification, and possibly transformation into executable code. This environment must be an open system in order to be able to integrate future tools. Of course we do not expect that at the end of our project there will be a complete development environment. Consequently we only intend to implement some important parts of such a system.

Acknowledgement

Special thanks to our colleagues G. Denker, H.-D. Ehrich, N. Vlachantonis (who are also involved in the KorSo project), and to P. Löhr-Richter for fruitful and encouraging discussions. The efforts of our colleagues T. Hartmann, R. Jungclaus, and G. Saake were the starting point for TROLL *light*. Furthermore, the comments and questions of our project partners from TU Berlin and FZI Karlsruhe and of A. Sernadas have been helpful in improving this paper. The authors accept full responsibility for any error or deficiency in the paper.

References

[BBE+90] G. v. Bochmann, M. Barbeau, M. Erradi, L. Lecomte, P. Mondain-Monval, and N. Williams. Mondel: An Object-Oriented Specification Language. Département d'Informatique et de Recherche Opérationnelle, Publication 748, Université de Montréal, 1990.

[Bee90] C. Beeri. A Formal Approach to Object-Oriented Databases. *Data & Knowledge Engineering*, 5(4):353–382, 1990.

[BFG+92] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, and K. Stølen. The Requirement and Design Specification Language SPECTRUM — An Informal Introduction (Version 0.3). Technical Report TUM–I9140, Technische Universität München, 1992.

[BHL90] D. Bjorner, C.A.R. Hoare, and H. Langmaack, editors. *VDM'90: VDM and Z — Formal Methods in Software Development.* Springer LNCS series, Vol. 428, 1990.

[Bre91] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments.* Springer LNCS Series, 1991.

[CHJ86] B. Cohen, W.T. Harwood, and M.I. Jackson. *The Specification of Complex Systems.* Addison-Wesley, Reading (MA), 1986.

[CSS89] J.-F. Costa, A. Sernadas, and C. Sernadas. OBL-89 Users Manual (Version 2.3). Internal report, INESC, Lisbon, 1989.

[DO91] O.-J. Dahl and O. Owe. Formal Development with ABEL. Technical Report 159, University of Oslo, 1991.

[EGH+92] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr-Richter, G. Saake, and H.-D. Ehrich. Conceptual modelling of database applications using an extended ER model. *Data & Knowledge Engineering, North-Holland*, 9(2):157–204, 1992.

[EGL89] H.-D. Ehrich, M. Gogolla, and U.W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen – Eine Einführung in die Theorie.* Teubner, Stuttgart, 1989.

[EGS90] H.-D. Ehrich, J. A. Goguen, and A. Sernadas. A Categorial Theory of Objects as Observed Processes. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages (Proc. REX/FOOL Workshop, Noordwijkerhood (NL))*, pages 203–228. Springer LNCS series, Vol. 489, 1990.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer, Berlin, 1985.

[EM90] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Modules and Constraints*. Springer, Berlin, 1990.

[Gab91] P. Gabriel. The Object-Based Specification Language Π: Concepts, Syntax, and Semantics. In *Proc. 8th Workshop on Specification of Abstract Data Types*. Springer LNCS series, 1991.

[GH91] M. Gogolla and U. Hohenstein. Towards a Semantic View of an Extended Entity-Relationship Model. *ACM Trans. on Database Systems*, 16(3):369–416, 1991.

[GW91] J.A. Goguen and D. Wolfram. On Types and FOOPS. In Meersman et al. [MKK91], pages 1–22.

[HE92] U. Hohenstein and G. Engels. SQL/EER — Syntax and Semantics of an Entity-Relationship-Based Query Language. *Information Systems*, 17(3):209–242, 1992.

[HK87] R. Hull and R. King. Semantic Database Modelling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs (NJ), 1985.

[JSH91] R. Jungclaus, G. Saake, and T. Hartmann. Language Features for Object-Oriented Conceptual Modeling. In T.J. Teorey, editor, *Proc. 10th Int. Conf. on Entity-Relationship Approach*, pages 309–324. E/R Institute, 1991.

[JSHS91] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The TROLL Language. Informatik-Bericht 91–04, Technische Universität Braunschweig, 1991.

[KL89] M. Kifer and G. Lausen. F-Logic: A Higher-Order Language for Reasoning About Objects, Inheritance, and Scheme. *ACM SIGMOD Record*, 18(2):134–146, 1989. Proc. ACM SIGMOD Conf. on Management of Data.

[KS91] G. Kappel and M. Schrefl. Using an Object-Oriented Diagram Technique for the Design of Information Systems. In H.G. Sol and K.M. Van Hee, editors, *Dynamic Modelling of Information Systems*, pages 121–164. Elsevier (North-Holland), 1991.

[LR89] C. Lécluse and P. Richard. Modeling Complex Structures in Object-Oriented Databases. In *Proc. 8th ACM Symp. Principles of Database Systems*, pages 360–368, 1989.

[LVVS90] E. Laenens, B. Verdonk, D. Vermeir, and D. Sacca. The LOCO Language: Towards an Integration of Logic and Object Oriented Programming. Report 90-09, University of Antwerpen, 1990.

[Mes90] J. Meseguer. A Logical Theory of Concurrent Objects. *ACM SIGPLAN Notices*, 25(10):101–115, 1990. Proc. OOPSLA.

[Mes92] J. Meseguer. Conditional Rewriting as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–156, 1992.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Springer, Berlin, 1980.

[MKK91] R.A. Meersman, W. Kent, and S. Khosla, editors. *Object-Oriented Databases: Analysis, Design & Construction (DS-4), Proc. IFIP WG 2.6 Working Conference, Windermere (UK) 1990.* North-Holland, 1991.

[Reg90] G. Reggio. Entities: An Institution for Dynamic Systems. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification*, pages 246–265. Springer LNCS series, Vol. 534, 1990.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* Springer, Berlin, 1985.

[Saa92] G. Saake. Objektorientierte Modellierung von Informationssystemen. Informatik-Skripten 28, Technische Universität Braunschweig, 1992.

[SE91] A. Sernadas and H.-D. Ehrich. What is an Object, after all? In Meersman et al. [MKK91], pages 39–70.

[SEC90] A. Sernadas, H.-D. Ehrich, and J.-F. Costa. From Processes to Objects. *The INESC Journal of Research and Development 1:1*, pages 7–27, 1990.

[SGG+91] C. Sernadas, P. Gouveia, J. Gouveia, A. Sernadas, and P. Resende. The Reification Dimension in Object-Oriented Database Design. In D. Harper and M. C. Norrie, editors, *Proc. Int. Workshop on Specification of Database Systems*, pages 275–299. Springer, 1991.

[SM92] S. Shlaer and S.J. Mellor. *Object Lifecycles: Modeling the World in States.* Yourdon Press computing series, Prentice-Hall, Englewood Cliffs (NJ), 1992.

[SRGS91] C. Sernadas, P. Resende, P. Gouveia, and A. Sernadas. In-the-large Object-Oriented Design of Information Systems. In F. Van Assche, B. Moulin, and C. Rolland, editors, *Proc. Object-Oriented Approach in Information Systems*, pages 209–232. North Holland, 1991.

[SSE87] A. Sernadas, C. Sernadas, and H.-D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In P.M. Stoecker and W. Kent, editors, *Proc. 13th Int. Conf. on Very Large Databases VLDB'87*, pages 107–116. VLDB Endowment Press, Saratoga (CA), 1987.

[SSG+91] A. Sernadas, C. Sernadas, P. Gouveia, P. Resende, and J. Gouveia. OBLOG — Object-Oriented Logic: An Informal Introduction. Technical report, INESC, Lisbon, 1991.

[TM87] W. Turski and T.S.E Maibaum. *The Specification of Computer Programs.* Addison-Wesley, Reading (MA), 1987.

[VB82] G.M.A Verheijen and J. Van Bekkum. NIAM: An Information Analysis Method. In T.W. Olle, H.G. Sol, and A.A. Verrijn-Stuart, editors, *Proc. "Information Systems Design Methodologies: A Comparative Review"*, pages 537–590, North-Holland, 1982.

[Wie91] R. Wieringa. Equational Specification of Dynamic Objects. In Meersman et al. [MKK91], pages 415–438.

[Win90] J.M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23:8–24, 1990.

[Wir90] M. Wirsing. Algebraic Specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 677–788. Elsevier, North-Holland, 1990.

A The Syntax

The symbols ::= |, [...], {...}, and <...> are meta-symbols. Vertical bars separate alternatives. Square brackets enclose optional items. Curly brackets denote repetition. Angled brackets enclose syntactic categories (non-terminal symbols). Type writer words and non-alphanumeric symbols enclosed in quote marks represent terminals.

We use the following abbreviations (let x be a non-terminal symbol):

```
<x_list>               ::= <x> {“,”<x>}
<x_seq>                ::= <x> {“;”<x>}
```

<x_id> is used for possible identifiers for items of type x.

```
<template_spec>        ::= TEMPLATE <template_id>
                              <template_sig> <template_body>
                           END TEMPLATE

<template_sig>         ::= [<datatypes_section>]
                           [<templates_section>]
                           [<subobj_section>]
                           [<attr_section>]
                           [<events_section>]

<template_body>        ::= [<constraints_section>]
                           [<valuation_section>]
                           [<derivation_section>]
                           [<interaction_section>]
                           [<behavior_section>]

<datatypes_section>    ::= DATA TYPES <datatype_id_list>“;”
<templates_section>    ::= TEMPLATES <template_id_list>“;”

<subobj_section>       ::= SUBOBJECTS <subobj_def_seq>“;”
<subobj_def>           ::= <subobj_name>“:”<objsort_id>
<subobj_name>          ::= <subobj_gen_id> [“(”<simple_par_list>“)”]
<simple_par>           ::= [<par_id>“:”] <simple_domain>
<simple_domain>        ::= <base_domain> | UNION“(”<base_domain_alt_list>“)”
<base_domain>          ::= <datasort_id> | <objsort_id>
<base_domain_alt>      ::= <alternative_id>“:”<base_domain>

<attr_section>         ::= ATTRIBUTES <attr_def_seq>“;”
<attr_def>             ::=   <attr_name>“:”<domain>
                           | DERIVED <derived_attr_id>“:”<domain>
<attr_name>            ::= <attr_gen_id> [“(”<base_par_list>“)”]
<base_par>             ::= [<par_id>“:”] <base_domain>
<domain>               ::= <base_domain> | SET“(”<domain>“)”
                                         | LIST“(”<domain>“)”
                                         | BAG“(”<domain>“)”
```

| TUPLE“(”<par_list>“)”
| UNION“(”<alternative_list>“)”
<par> ::= [<par_id>“:”] <domain>
<alternative> ::= <alternative_id>“:”<domain>

<events_section> ::= EVENTS <event_def_seq>“;”
<event_def> ::= [BIRTH | DEATH] <event_id> [“(”<par_list>“)”]

<constraints_section> ::= CONSTRAINTS <formula_seq>“;”

<valuation_section> ::= VALUATION <val_rule_seq>“;”
<val_rule> ::= [“{”<formula>“}”] “[”<event_descr>“]” <effect_list>
<event_descr> ::= <event_id> [“(”<var_id_list>“)”]
<effect> ::= <attr_term> “=” <term>
<attr_term> ::= <attr_gen_id> [“(”<actual_par_list>“)”]
<actual_par> ::= [<par_id>“:”] <term>

<derivation_section> ::= DERIVATION <derivation_rule_seq>“;”
<derivation_rule> ::= <derived_attr_id> “=” <term>

<interaction_section> ::= INTERACTION <interaction_def_seq>“;”
<interaction_def> ::= [“{”<formula>“}”] <source> “>>” <destination_list>
<source> ::= [<term>“.”] <event_descr>
<destination> ::= [<term>“.”] <event_term>
<event_term> ::= <event_id> [“(”<actual_par_list>“)”]

<behavior_section> ::= BEHAVIOR [<process_def_seq>“;”] <process>“;”
<process_def> ::= PROCESS <process_id> “=” <process>
<process> ::= “(”[<sequence> {“|”<sequence>}]“)”
<sequence> ::= <cond_event_list> {“->”<cond_event_list>}
“->” <end_of_sequence>
<end_of_sequence> ::= <process> | <process_id>
<cond_event> ::= [“{”<formula>“}”] <event_descr>

<term> and <formula> correspond to syntactic categories of an object calculus, which is an offspring of the EER calculus proposed in [GH91]. On the basis of this calculus SQL/EER, a query language for extended Entity-Relationship schemas, was developed [HE92]. A slightly modified subset of this language will satisfy our proposals.

<term> ::= <constant>
| <var_id>
| <prefixop>[“(”<term_list>“)”]
| <term> <infixop> <term>
| “(”<term>“)”
| “(”IF <formula> THEN <term> ELSE <term>“)”
| “(”SELECT <term_list> FROM <decl_list> [WHERE <formula>]“)”

```
<constant>        ::=  <integer_value> | <real_value> | <string_value>
                     | <bool_value> | EMPTY | BOTTOM | SELF
<integer_value>   ::= [“+”|“-”]<digits>
<real_value>      ::= [“+”|“-”]<digits>“.”<digits>[E[“+”|“-”]<digits>]
<digits>          ::= <digit>{<digit>}
<string_value>    ::= “"”{<char>}“"”
<bool_value>      ::= TRUE | FALSE

<prefixop>        ::= <datasort_op> | <objsort_op> | <sortexpr_op>
<objsort_op>      ::= <subobj_gen_id> | <attr_gen_id>
<sortexpr_op>     ::= CNT | SEL | PRJ | <par_id> | <alternative_id> | ...
                      SET | LIST | BAG | TUPLE | <alternative_id>“:” |
                      ADD | REMOVE | DELETE | ...
<infixop>         ::= “+” | “-” | “*” | “/” | MOD | DIV

<decl>            ::= <var_id> IN <term>

<formula>         ::=  <datasort_pred>[“(”<term_list>“)”]
                     | <term> <infixpred> <term>
                     | “(”<formula>“)”
                     | DEF“(”<term>“)”
                     | <formula> <connective> <formula>
                     | NOT <formula>
                     | <quantor> “(”<decl>“)” <formula>

<infixpred>       ::= “=” | “<>” | “>” | “>=” | “<” | “<=” | IN
<connective>      ::= AND | OR | IMPLIES
<quantor>         ::= FORALL | EXISTS
```